The VLDB Journal Manuscript-Nr.
(will be inserted by hand later)

Distance-based outliers: algorithms and applications

Edwin M. Knorr[1], Raymond T. Ng[1] and Vladimir Tucakov[2]

[1] Department of Computer Science, University of British Columbia, Vancouver, BC, V6T 1Z4, Canada
[2] Point Grey Research Inc., 101 - 1847 West Broadway, Vancouver, BC, V6J 1Y6, Canada

Received: Editor's Date1 / Accepted: Editor's Date2

Abstract. This paper deals with finding outliers (exceptions) in large, multidimensional datasets. The identification of outliers can lead to the discovery of truly unexpected knowledge in areas such as electronic commerce, credit card fraud, and even the analysis of performance statistics of professional athletes. Existing methods that we have seen for finding outliers can only deal efficiently with two dimensions/attributes of a dataset. In this paper, we study the notion of *DB*- (*Distance-Based*) outliers. Specifically, we show that: (i) outlier detection can be done *efficiently* for *large* datasets, and for *k-dimensional* datasets with large values of k (e.g., $k \geq 5$); and (ii), outlier detection is a *meaningful* and important knowledge discovery task.

First, we present two simple algorithms, both having a complexity of $O(kN^2)$, k being the dimensionality and N being the number of objects in the dataset. These algorithms readily support datasets with many more than two attributes. Second, we present an optimized cell-based algorithm that has a complexity that is linear with respect to N, but exponential with respect to k. We provide experimental results indicating that this algorithm significantly outperforms the two simple algorithms for $k \leq 4$. Third, for datasets that are mainly disk-resident, we present another version of the cell-based algorithm that guarantees at most three passes over a dataset. Again, experimental results show that this algorithm is by far the best for $k \leq 4$. Finally, we discuss our work on three real-life applications, including one on spatio-temporal data (e.g., a video surveillance application), in order to confirm the relevance and broad applicability of DB-outliers.

Key words: Outliers – Data Mining – Data Mining Applications – Algorithms – Exceptions

1 Introduction

Knowledge discovery tasks can be classified into four general categories: (a) dependency detection, (b) class identification, (c) class description, and (d) exception/outlier detection. The first three categories of tasks correspond to patterns that apply to many objects, or to a large percentage of objects, in the dataset. Most research in data mining—such as association rules [1], classification [5], and data clustering [10, 24]—belongs to these three categories. The fourth category, in contrast, focuses on a very small percentage of data objects, which are often ignored or discarded as noise. For example, some existing algorithms in machine learning and data mining have considered outliers, but only to the extent of tolerating them in whatever the algorithms are supposed to do [10, 24].

It has been said that, "One person's noise is another person's signal." Indeed, for some applications, the rare events are often more interesting than the common ones, from a knowledge discovery standpoint. Sample applications include the detection of credit card fraud and the monitoring of criminal activities in electronic commerce. For example, in Internet commerce or smart card applications, we expect many low-value transactions to occur. However, it is the exceptional cases—exceptional perhaps in monetary amount, type of purchase, timeframe, location, or some combination thereof—that may interest us, either for fraud-detection or marketing reasons.

1.1 Related work

Most of the existing work on outlier detection lies in the field of statistics [3, 17]. While there is no single, generally accepted, formal definition of an outlier, Hawkins' definition captures the spirit: "an outlier is an observation that deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism" [17]. Accordingly, over one hundred discordancy/outlier tests have been developed for different circumstances, depending on: (i) the data distribution, (ii) whether or not the distribution parameters (e.g., mean and variance) are known, (iii) the number of expected outliers, and even (iv) the types of expected outliers (e.g., upper or lower outliers in an ordered sample) [3]. However, all of those tests suffer from the following two serious problems. First, almost of them are univariate (i.e., single attribute). This restriction makes them unsuitable for multidimensional datasets. Second, all of them are distribution-based. In numerous situations where we do not know whether a particular attribute follows a normal distribution, a gamma distribution, and so on, we have to perform extensive testing to find a distribution that fits the attribute.

To help improve the situation, some methods in computational statistics have been developed, which can be best described as depth-based. Based on some definition of depth, data objects are organized in layers in the data space, with the expectation that shallow layers are more likely to contain outlying data objects than are the deep layers. *Peeling* and *depth contours* are two different notions of depth studied in [25, 26]. These depth-based methods avoid the aforementioned problem of distribution fitting, and conceptually allow multidimensional data objects to be processed. However, in practice, the computation of k-dimensional layers relies on the computation of k-dimensional convex hulls. Because the lower bound complexity of computing a k-dimensional convex hull for N data objects is $\Omega(N^{\lceil \frac{k}{2} \rceil})$, depth-based methods are not expected to be practical for more than 4 dimensions for large datasets. In fact, existing depth-based methods only give acceptable performance for $k \leq 2$ [26].

Arning, et al. [2] search a dataset for implicit redundancies, and extract data objects called *sequential exceptions* that maximize the reduction in Kolmogorov complexity. This notion of outliers is very different from the aforementioned statistical definitions of outliers. As will be seen shortly, it is also very different from the notion of outliers considered here, primarily because there is not an associated notion of distance and similarity measure between objects.

A few clustering algorithms, such as CLARANS [24], DBSCAN [10], and BIRCH [32], are developed with exception-handling capabilities. However, their main objective is to find clusters in the dataset. As such, their notions of outliers are defined indirectly through the notion of clusters, and they are developed only to optimize clustering, but not to optimize outlier detection.

Finally, we comment on two contemporary studies for identifying exceptions in large, multidimensional datasets. The techniques employed are significantly different from those of DB-outliers, yet the spirit is similar: the anomalies (outliers) that are identified are "intuitively surprising".

The work by Chakrabarti, et al. deals with the complex problem of finding surprising temporal patterns in market basket data [8]. The authors denote the level of surprise by the number of bits used to encode a market basket sequence whose items possess statistically strong variation of correlation along time. Sarawagi, et al. allow a user to search for anomalies in OLAP data cubes [28]. The authors define an anomaly as a data value that is significantly different from the value anticipated based on a statistical model. A user navigates through the data cube, visually identifying interesting paths. This interactive approach involves the user in the discovery process. Our work on distance-based outliers also involves the user in the discovery process, as evidenced by user-supplied parameters p and D in the formal definition of DB-outliers given in the next section.

1.2 Contributions of this paper

The two main objectives/results of this paper are as follows:

I. Outlier detection can be done *efficiently* for *large* datasets, and for *k-dimensional* datasets with large values of k (e.g., $k \geq 5$).
II. Outlier detection is a *meaningful* and important knowledge discovery task.

With respect to the first objective, we study the notion of *Distance-Based* outliers defined as follows:

An object O in a dataset T is a $DB(p, D)$-outlier if at least fraction p of the objects in T lies greater than distance D from O.

With this notion of outliers, we present two algorithms. The first algorithm has a complexity of $O(kN^2)$, which is computationally far superior to the depth-based outliers for larger values of k. The second algorithm, has a complexity of $O(c^k + N)$, where c is some small constant dependent on both $\sqrt{k}$ and (roughly) the average number of cells into which each dimension (attribute) has been partitioned. Since this algorithm is exponential on k, it is recommended only for small values of k.

With respect to the second objective, we first give an overview of some of the applications on which we have had first-hand experience. Then we describe in greater detail one particular case study or application that uses surveillance video to identify suspicious behavior of people in public places. The specific purpose of the study is to develop and evaluate a method for outlier detection in the domain of 2-D motion tracking. The task at hand is the motion tracking of people walking through an open space such a building atrium or a large hallway. People generally take similar paths as they walk from point A to point B in a building. Using distance-based outlier detection algorithms, our system identifies paths that are unusual, and brings the paths to the attention of the human user.

2 Properties of distance-based outliers: generalizations of discordancy tests

Definition 1. $DB(p, D)$ *unifies*[1] or *generalizes* another definition *Def* for outliers, if there exist specific values p_0, D_0 such that: object O is an outlier according to *Def* iff O is a $DB(p_0, D_0)$-outlier. □

For a normal distribution, outliers can be considered to be points that lie 3 or more standard deviations (i.e., $\geq 3\sigma$) from the mean μ [13].

[1] DB-outliers are called *unified* outliers in our preliminary work [22].

Definition 2. Let T be a normally distributed random variable with mean μ and standard deviation σ. Define Def_{Normal} as follows: $t \in T$ is an outlier iff $\frac{t-\mu}{\sigma} \geq 3$ or $\frac{t-\mu}{\sigma} \leq -3$. □

Lemma 1. $DB(p, D)$ unifies Def_{Normal} with $p_0 = 0.9988$ and $D_0 = 0.13\sigma$, that is, t is an outlier according to Def_{Normal} iff t is a $DB(0.9988, 0.13\sigma)$-outlier.

Proofs of all the lemmas in this section have already been documented [22]. Note that if the value 3σ in Def_{Normal} is changed to some other value, such as 4σ, the above lemma can easily be modified with the corresponding p_0 and D_0 to show that $DB(p, D)$ still unifies the new definition of Def_{Normal}. The same general approach applies to a Student t-distribution, which has fatter tails than a normal distribution. The principle of using a tail to identify outliers also applies to a Poisson distribution.

Definition 3. Define Def_{Poisson} as follows: t is an outlier in a Poisson distribution with parameter $\mu = 3.0$ iff $t \geq 8$. □

Lemma 2. $DB(p, D)$ unifies Def_{Poisson} with $p_0 = 0.9892$ and $D_0 = 1$. □

Finally, for a class of regression models, we can define an outlier criterion $Def_{\text{Regression}}$, and show that $DB(p, D)$ unifies $Def_{\text{Regression}}$ [22]. Consider the simple linear regression model given by the equation $y = \alpha + \beta x$. Suppose dataset T consists of observations of the form (x_i, y_i) for $i = 1, 2, \ldots, N$, that are fitted to this model. We denote the residual or error, that is, the difference between the observed value and the fitted value for the i'th observation, by e_i in the equation $y_i = \alpha + \beta x_i + e_i$. One possible way of identifying outliers is by finding outlying residuals. An outlier among residuals can be considered to be a residual that is far greater than the rest in absolute value, and which lies at least 3 standard deviations from the mean of the residuals [9]. If we make the simplifying assumption (as some authors do) that residuals are independent and follow a normal distribution, then we can easily identify outliers using the same approach demonstrated above.

3 Simple algorithms for finding all $DB(p, D)$-outliers

3.1 Index-based algorithms

Let N be the number of objects in the input dataset T, and let F be the underlying distance function that gives the distance between any pair of objects in T. For an object O, the D-neighbourhood of O contains the set of objects $Q \in T$ that are within distance D of O (i.e., $\{Q \in T \mid F(O, Q) \leq D\}$). The fraction p is the minimum fraction of objects in T that must be *outside* the D-neighbourhood of an outlier. For simplicity of discussion, let M be the maximum number of objects within the D-neighbourhood of an outlier, that is, $M = N(1 - p)$.

From the formulation above, it is obvious that given p and D, the problem of finding all $DB(p, D)$-outliers can be solved by answering a nearest neighbour or range query centered at each object O. More specifically, based on a standard multidimensional indexing structure, we can execute a range search with radius D for each object O. Once there are at least $(M + 1)$ neighbours in the D-neighbourhood, we stop the search and declare O a non-outlier. Otherwise, we report O as an outlier.

Analyses of multidimensional indexing schemes [18] reveal that, for variants of R-trees [15], k-d trees [4, 27], and X-trees [6], the lower bound complexity for a range search is $\Omega(N^{1-1/k})$. As k increases, a range search quickly reduces to $O(N)$, giving at best a constant time improvement reflecting sequential search. Thus, the above procedure for finding all $DB(p, D)$-outliers has a worst case complexity of $O(k N^2)$. Two points are worth noting:

- Compared to the depth-based approaches, which have a lower bound complexity of $\Omega(N^{\lceil \frac{k}{2} \rceil})$, DB-outliers scale much better with dimensionality. The framework of DB-outliers is applicable and computationally feasible for datasets that have many attributes (i.e., $k \geq 5$). This is a significant improvement on the current state-of-the-art, where existing methods can only realistically deal with two attributes [26].
- The above analysis only considers search time. When it comes to using an index-based algorithm, most often for the kinds of data mining applications under consideration, it is a very strong assumption that the right index exists. In other words, a huge hidden cost to an index-based algorithm is the effort to build the index in the first place. As will be shown in Sect. 6, the index building cost alone, even without counting the search cost, almost always renders the index-based algorithms non-competitive.

3.2 A nested-loop algorithm

To avoid the cost of building an index for finding all $DB(p, D)$-outliers, algorithm NL shown in Fig. 1 uses a block-oriented, nested-loop design. Assuming a total buffer size of B% of the dataset size, the algorithm divides the entire buffer space into two halves, called the *first* and *second* arrays in Fig. 1. It reads the dataset into the arrays, and directly computes the distance between each pair of objects or tuples.[2] For each object t in the first array, a count of its D-neighbours is maintained.[3] Counting stops for a particular tuple whenever the number of D-neighbours exceeds M.

[2] From here on in, we use the terms *object* and *tuple* interchangeably.

[3] The version of algorithm NL considered in this paper only has space for a counter for each tuple in the first array. A variant is to assume that the algorithm also maintains a counter for each tuple in the second array—which effectively means for every block and tuple in the dataset. Because this variant requires much more space than algorithm NL, we do not pursue it further, but most of our comments concerning algorithm NL can easily be generalized for the variant.

Algorithm NL

1. Fill the first array (of size $\frac{B}{2}$% of the dataset) with a block of tuples from T.
2. For each tuple t_i in the first array, do:
   a. $count_i \leftarrow 0$
   b. For each tuple t_j in the first array, if dist(t_i, t_j) $\leq D$:
      Increment $count_i$ by 1. If $count_i > M$, mark t_i as a non-outlier and proceed to next t_i.
3. While blocks remain to be compared to the first array, do:
   a. Fill the second array with another block (but save a block which has never served as the first array, for last).
   b. For each unmarked tuple t_i in the first array do:
      For each tuple t_j in the second array, if dist(t_i, t_j) $\leq D$:
         Increment $count_i$ by 1. If $count_i > M$, mark t_i as a non-outlier and proceed to next t_i.
4. For each unmarked tuple t_i in the first array, report t_i as an outlier.
5. If the second array has served as the first array anytime before, stop; otherwise, swap the names of the first and second arrays and goto step 2.

Fig. 1. Pseudo-code for algorithm NL

Consider algorithm NL with 50% buffering, and denote the 4 logical blocks of the dataset by A, B, C, D, with each block containing $\frac{1}{4}$ of the dataset. Let us follow the algorithm, filling the arrays in the following order, and comparing:

1. A with A, then with B, C, D—for a total of 4 block reads;
2. D with D (no read required), then with A (no read), B, C—for a total of 2 block reads;
3. C with C, then with D, A, B—for a total of 2 blocks reads, and
4. B with B, then with C, A, D—for a total of 2 block reads.

Thus, a grand total of 10 blocks are read, amounting to $\frac{10}{4} = 2.5$ passes over the entire dataset. Later, in Sect. 5.3, we compute the number of passes required in the general case.

As compared to the index-based algorithms, algorithm NL avoids the explicit construction of any indexing structure. It is easy to see that its complexity is $O(kN^2)$. In the following two sections, we present two versions of a cell-based algorithm that has a complexity linear with respect to N, but exponential with respect to k. This cell-based algorithm is therefore intended only for small values of k. The key idea is to gain efficiency by using cell-by-cell processing, instead of tuple-by-tuple processing, thereby avoiding the N^2 complexity term.

4 A cell-based approach

To illustrate the idea and the various optimizations of the cell-based algorithm, we first present a *simplified* version of the algorithm. This version assumes that both the multidimensional cell structure and the entire dataset fit into main memory. For ease of presentation, we begin with 2-D, and then proceed to k-D. In the next section, we present the full version of the algorithm that can handle disk-resident datasets.

4.1 Cell structure and properties in 2-D

Suppose our data objects are 2-D. For our cell-based search strategy for finding all $DB(p, D)$-outliers in 2-D, we quantize each of the data objects into a 2-D space that has been partitioned into cells or squares of length $l = \frac{D}{2\sqrt{2}}$. Let $C_{x,y}$ denote the cell that is at the intersection of row x and column y. The Layer 1 (L_1) neighbours of $C_{x,y}$ are the immediately neighbouring cells of $C_{x,y}$, defined in the usual sense, that is,

$$L_1(C_{x,y}) = \{C_{u,v} \mid u = x \pm 1,\ v = y \pm 1,\ C_{u,v} \neq C_{x,y}\}. \quad (1)$$

A typical cell (except for any cell on the boundary of the cell structure) has 8 L_1 neighbours.

Property 1. Any pair of objects within the same cell is at most distance $\frac{D}{2}$ apart. □

Property 2. If $C_{u,v}$ is an L_1 neighbour of $C_{x,y}$, then any object $P \in C_{u,v}$ and any object $Q \in C_{x,y}$ are at most distance D apart. □

Property 1 is valid because the length of a cell's diagonal is $\sqrt{2}l = \sqrt{2}\frac{D}{2\sqrt{2}} = \frac{D}{2}$. Property 2 is valid because the distance between any pair of objects in the two cells cannot exceed twice the length of a cell's diagonal. We will see that these two properties are useful in ruling out many objects as outlier candidates. The Layer 2 (L_2) neighbours of $C_{x,y}$ are those additional cells within 3 cells of $C_{x,y}$, that is,

$$L_2(C_{x,y}) = \{C_{u,v} \mid u = x \pm 3,\ v = y \pm 3, C_{u,v} \notin L_1(C_{x,y}),\ C_{u,v} \neq C_{x,y}. \quad (2)$$

A typical cell (except for any cell on or near a boundary) has $7^2 - 3^2 = 40$ L_2 cells. Note that Layer 1 is 1 cell thick and Layer 2 is 2 cells thick. L_2 was chosen in this way to satisfy the following property.

Property 3. If $C_{u,v}$ is neither an L_1 nor an L_2 neighbour of $C_{x,y}$, and $C_{u,v} \neq C_{x,y}$, then any object $P \in C_{u,v}$ and any object $Q \in C_{x,y}$ must be at least distance D apart. □

Since the combined thickness of L_1 plus L_2 is 3 cells, the distance between P and Q must exceed $3l = \frac{3D}{2\sqrt{2}} > D$.

Property 4. (a) If there are $> M$ objects in $C_{x,y}$, *none* of the objects in $C_{x,y}$ is an outlier.
(b) If there are $> M$ objects in $C_{x,y} \cup L_1(C_{x,y})$, *none* of the objects in $C_{x,y}$ is an outlier.
(c) If there are $\leq M$ objects in $C_{x,y} \cup L_1(C_{x,y}) \cup L_2(C_{x,y})$, *every* object in $C_{x,y}$ is an outlier. □

Properties 4(a) and 4(b) are direct consequences of Properties 1 and 2, and 4(c) is due to Property 3.

Algorithm FindAllOutsM

1. For $q \leftarrow 1, 2, \ldots m$, $\quad Count_q \leftarrow 0$
2. For each object P, map P to an appropriate cell C_q, store P, and increment $Count_q$ by 1.
3. For $q \leftarrow 1, 2, \ldots, m$, $\quad$ if $Count_q > M$, label C_q *red*.
4. For each *red* cell C_r, label each of the L_1 neighbours of C_r *pink*, provided the neighbour has not already been labelled *red*.
5. For each non-empty *white* (i.e., uncoloured) cell C_w, do:
   a. $Count_{w2} \leftarrow Count_w + \sum_{i \in L_1(C_w)} Count_i$
   b. If $Count_{w2} > M$, label C_w *pink*.
   c. else
      1. $Count_{w3} \leftarrow Count_{w2} + \sum_{i \in L_2(C_w)} Count_i$
      2. If $Count_{w3} \leq M$, mark all objects in C_w as outliers.
      3. else for each object $P \in C_w$, do:
         i. $Count_P \leftarrow Count_{w2}$
         ii. For each object $Q \in L_2(C_w)$, if $dist(P, Q) \leq D$: Increment $Count_P$ by 1. If $Count_P > M$, P cannot be an outlier, so goto 5(c)(3).
         iii. Mark P as an outlier.

Fig. 2. Pseudo-code for algorithm FindAllOutsM

4.2 Algorithm FindAllOutsM for memory-resident datasets

Figure 2 presents algorithm FindAllOutsM to detect all $DB(p, D)$-outliers in memory-resident datasets. Later, in Sect. 5.2, we present an enhanced algorithm to handle disk-resident datasets.

Step 2 of algorithm FindAllOutsM quantizes each tuple to its appropriate cell. Step 3 labels all cells containing $> M$ tuples, *red*. This corresponds to Property 4(a). Cells that are L_1 neighbours of a *red* cell are labelled *pink* in step 4, and cannot contain outliers because of Property 4(b). Other cells satisfying Property 4(b) are labelled *pink* in step 5(b). Finally, in step 5(c)(2) of the algorithm, cells satisfying Property 4(c) are identified.

To summarize, all of the properties mentioned in Sect. 4.1 are used to help determine outliers and non-outliers on a cell-by-cell basis, rather than on an object-by-object basis. This helps reduce execution time significantly because we quickly rule out a large number of objects that cannot be outliers. For cells not satisfying any of Properties 4(a)-(c), we must resort to object-by-object processing for the objects which have been quantized to those cells. Such cells are denoted as *white* cells (C_w). In step 5(c)(3) of algorithm FindAllOutsM, each object $P \in C_w$ may have to be compared with every object Q lying in a cell that is an L_2 neighbour of C_w in order to determine how many Q's are inside the D-neighbourhood of P. As soon as the number of D-neighbours exceeds M, P is declared a non-outlier. If, after examining all Q's, the count remains $\leq M$, then P is an outlier.

4.3 Complexity analysis: the 2-D case

Let us analyze the complexity of algorithm FindAllOutsM for the 2-D case. Step 1 takes $O(m)$ time, where $m \ll N$ is the total number of cells. Steps 2 and 3 take $O(N)$ and $O(m)$ time, respectively. Since M is the maximum number of objects that can appear in the D-neighbourhood of an outlier, there are at most $\frac{N}{M}$ *red* cells. Thus, step 4 takes $O(\frac{N}{M})$ time. The time complexity of step 5 is the most complicated. In the worst case, (i) no cell is labelled *red* or *pink* in the previous steps, and (ii) step 5(c) is necessary for all cells. If no cell is coloured, then each cell contains at most M objects. Thus, in step 5(c), each of the objects in a cell can require the checking of up to M objects in each of the 40 L_2 neighbours of the cell; therefore, $O(40M^2)$ time is required for each cell. Hence, step 5 takes $O(mM^2)$ time. Since, by definition, $M = N(1-p)$, we equate $O(mM^2)$ to $O(mN^2(1-p)^2)$. In practice, we expect p to be extremely close to 1, especially for large datasets, so $O(mN^2(1-p)^2)$ can be approximated by $O(m)$. Thus, the time complexity of algorithm FindAllOutsM in 2-D is $O(m+N)$. Note that this complexity figure is very conservative, because in practice, we expect many *red* and *pink* cells. As soon as this happens, there are fewer object-to-object comparisons. Thus, step 5(c) becomes less dominant, and the algorithm requires less time. In Sect. 6.2, we show experimental results on the efficiency of algorithm FindAllOutsM.

4.4 Generalization to higher dimensions

When moving from $k = 2$ dimensions to $k > 2$, algorithm FindAllOutsM requires only one change to incorporate a general k-D cell structure. That change involves the cell length l. Recall that in 2-D, $l = \frac{D}{2\sqrt{2}}$. Since the length of a diagonal of a k-D hypercube/cell of length l is $\sqrt{k}l$, the length l in a k-D setting must be changed to:

$$l = \frac{D}{2\sqrt{k}} \tag{3}$$

to preserve Properties 1 and 2.

Although the following comments do not change algorithm FindAllOutsM, an understanding of which cells appear in Layer 2 is important in correctly applying the algorithm. First, we note that the L_1 neighbours of cell $C_{x_1,\ldots,x_k}$ are:

$$\begin{aligned} L_1(C_{x_1,\ldots,x_k}) = \{&C_{u_1,\ldots,u_k} \mid \\ &u_i = x_i \pm 1 \;\; \forall\, 1 \leq i \leq k,\;\; C_{u_1,\ldots,u_k} \neq C_{x_1,\ldots,x_k}\} \end{aligned} \tag{4}$$

which generalizes the definition given in Eq. (1). However, to preserve Property 3, the definition of L_2 neighbours needs to be modified. Specifically, since $l = \frac{D}{2\sqrt{k}}$, Layer 2 needs to be thicker than it is for $k = 2$. Let x denote the thickness of Layer 2. Then, the combined thickness of Layers 1 and 2 is $x + 1$. So, for Property 3 to hold, we require that $(x+1)l > D$; consequently, we pick x as $\lceil 2\sqrt{k} - 1 \rceil$. The L_2 neighbours of $C_{x_1,\ldots,x_k}$ are therefore:

$$\begin{aligned} L_2(C_{x_1,\ldots,x_k}) = \{&C_{u_1,\ldots,u_k} \mid \\ &u_i = x_i \pm \lceil 2\sqrt{k} \rceil \;\; \forall\, 1 \leq i \leq k, \\ &C_{u_1,\ldots,u_k} \notin L_1(C_{x_1,\ldots,x_k}),\;\; C_{u_1,\ldots,u_k} \neq C_{x_1,\ldots,x_k}\} \end{aligned} \tag{5}$$

which generalizes Eq. (2). We have the following result to ascertain the correctness of the k-D cell structure.

Lemma 3. As formalized in Eqs. (3), (4), and (5), the k-D cell structure preserves Properties 1 to 4. $\square$

The proof of the lemma is straightforward. The consequence of the lemma is that algorithm FindAllOutsM can be applied to a k-D cell structure without any change.

For any value of $k > 2$, while FindAllOutsM requires no change, the complexity of the algorithm does change. Note that the time complexities of steps 1 to 4 in algorithm FindAllOutsM remain the same. However, m is now exponential with respect to k, and may not necessarily be much less than N. Also, the complexity of step 5 is no longer $O(m)$, but $O(m(2\lceil 2\sqrt{k}\rceil + 1)^k) \approx O(c^k)$, where c is some constant depending on $\sqrt{k}$ and on $m^{1/k}$ (which roughly corresponds to the number of cells along each dimension). Thus, the complexity of the entire algorithm is $O(c^k + N)$.

While this complexity figure represents a worst case scenario, the question to ask is how efficient algorithm FindAllOutsM is in practice for the general k-D case. We defer the answer to Sect. 6.4, but make the following preliminary comments. First, for the identification of "strong" outliers, the number of outliers to be found is intended to be small. This is achieved by having a large value of D, and a value of p very close to unity. A large value of D corresponds to a small number of cells along each dimension. Thus, the constant c is small, although it is greater than 1. Second, for values of p very close to unity, M is small, implying that there will be numerous *red* and *pink* cells. This means that the savings realized by skipping *red* and *pink* cells is enormous, and that the number of objects requiring pairwise comparisons is relatively small. Finally, as the dimensionality k increases, there is usually an increasing number of empty cells. Thus, in practice, the number of non-empty cells is much smaller than m. This makes the actual performance of the algorithm much more attractive than the complexity figure suggests.

4.5 Generalization to other distance functions

Instead of using Euclidean distance (the L_2-norm) for our distance function, we can generalize to the L_P-norm.[4] Specifically, the cell length l must be changed from $l = \frac{D}{2\sqrt{k}}$ (cf: Sect. 4.4) to $l = \frac{D}{2\sqrt[P]{k}}$ to preserve Properties 1 and 2 listed in Sect. 4.1. While the thickness of Layer 1 remains at one cell, and the definition of Layer 1 neighbours remains unchanged, the thickness of Layer 2 must be changed for Property 3 to hold. In particular, if the thickness of Layer 2 is denoted by x, then we require that $(x+1)l > D$; consequently, we pick x as $\lceil 2\sqrt[P]{k} - 1\rceil$. Therefore, the Layer 2 neighbours of cell $C_{x_1,\ldots,x_k}$ in k dimensions are now:

$$L_2(C_{x_1,\ldots,x_k}) = \{C_{u_1,\ldots,u_k} \mid u_i = x_i \pm \lceil 2\sqrt[P]{k}\rceil \ \forall\, 1 \le i \le k, \ C_{u_1,\ldots,u_k} \notin L_1(C_{x_1,\ldots,x_k}), \ C_{u_1,\ldots,u_k} \ne C_{x_1,\ldots,x_k}\} \quad (6)$$

which generalizes Eq. (5) in Sect. 4.4. Because of these changes, Properties 1 to 4 listed in Sect. 4.1 are preserved.

For example, Manhattan distance is given by the L_1-norm. The distance between two k-dimensional points x and y is given by: $dist(x,y) = |x_1 - y_1| + |x_2 - y_2| + \ldots + |x_k - y_k|$. Using the Manhattan distance function, the cell length becomes $l = \frac{D}{2k}$, and the Layer 2 neighbours of a given cell are now at offset $\pm\lceil 2k\rceil$ instead of at offset $\pm\lceil 2\sqrt[P]{k}\rceil$ shown in Eq. (6). Thus, the Layer 2 cell boundaries are no longer delimited by a hyperrectangle, but by a multidimensional "staircase" function.

Although we have dealt with Euclidean distance in this paper, we are researching ways to incorporate user-defined distance functions—including statistical distance functions, which account for variability among attributes [21]. Consider the case of looking for outliers in an application that has blood pressure as one of its attributes, and age as another attribute. Since the units of measurement are significantly different, it is obvious that the two attributes cannot be compared fairly to each other. Also, if all ages in the application are for patients aged 40-44 years, for example, then there will be little variation in the age attribute, but possibly substantial variations in the blood pressure attribute. This example motivates the need for statistical distance functions. One way to proceed is to divide each attribute by its sample standard deviation to obtain "standardized" attributes, and then apply the standard Euclidean distance function. (Standardization and the L_P-norm are orthogonal issues.) Thus, in k-dimensions, all points P that are a constant squared distance from a point Q lie on a hyper-ellipsoid centered at Q. There are also statistical distance functions to account for situations in which various attributes are correlated. We defer an in-depth treatment of outliers and statistical distance functions to future work.

4.6 Discussion: determination of initial values for p and D

Let us begin by stating that there is no universally correct value for p or D. Our idea is to permit the user to change the values of p and D during the search for outliers. The performance results shown in Sect. 6 will show that DB-outliers can be computed relatively quickly for various datasets. Hence, performing several iterations is often acceptable during exploration. Our approach to mining outliers incorporates the user in the knowledge discovery process. This is intensional because we believe that only a user who is familiar with the domain at hand can determine whether or not the tuples returned are meaningful. Directly related to this notion of meaningfulness is the user's choice of p and D. Let us now consider some heuristics for choosing appropriate values for these two parameters.

[4] The term L_2-norm is not related in any way to the Layer 2 acronym (L_2) which we have been using in this paper. The L_P-norm defines the distance between any two k-dimensional points x and y by: $dist(x,y) = \left[(|x_1 - y_1|)^P + (|x_2 - y_2|)^P + \ldots + (|x_k - y_k|)^P\right]^{1/P}$.

Intuitively, an outlier occurs relatively infrequently, and it is therefore reasonable to select a value of p very close to unity. Suppose we have a dataset of size $N = 1000$, and that p=0.995. Given the formula $M = N(1 - p)$, where M is the maximum number of tuples in the D-neighbourhood of an outlier Q, this means that only M=5 of the 1000 tuples in the dataset can lie inside of Q's D-neighbourhood. If D is sufficiently large, very few, if any, outliers may occur. Of course, for a sufficiently small value of D, it is possible that *every* tuple is an outlier. Our experience has been that for larger values of N, p should be closer to unity. For example, p=0.995 may suffice when $N \approx 10^3$, but may be far too small when $N \approx 10^6$. In the latter case, a value like p=0.99995 may be more appropriate.

One method of determining a reasonable starting value for D given some fixed p close to unity, and in the absence of other information, is via sampling. Sampling can be tricky. Due to the infrequent and unpredictable occurrence of outliers, it is difficult to sample and provide a high degree of confidence for D. We plan to address sampling in future work. At the very least, sampling is likely to reduce the number of iterations by providing a reasonable starting value for D. The user can increase or decrease D after the first iteration.

Finally, we suggest that the value of D not have more significant digits than those of the attribute values in the dataset.

5 $DB(p, D)$-outliers in large, disk-resident datasets

In the last section, we presented a cell-based algorithm that was simplified for memory-resident datasets. Here, we extend the simplified version to handle disk-resident datasets. This new version preserves the property of being linear with respect to N. It also provides the guarantee that no more than three, if not two, passes over the dataset are required. We begin with a motivating example.

5.1 Handling large disk-resident datasets: an example

In handling large, disk-resident datasets, the goal is to minimize the number of page reads or passes over the datasets. In the cell-based algorithm, there are two places where page reads are needed:

- The *initial-mapping* phase
  In step 2 of algorithm FindAllOutsM, each object is mapped to an appropriate cell. This step requires one pass over the dataset, and is unavoidable.
- The *object-pairwise* phase
  In step 5(c)(3), for each object P in a *white* cell C_w, each object Q in a cell that is an L_2 neighbour of C_w needs to be read to conduct the object-by-object distance calculation. Because we do not assume that objects mapped to the same cell, or to nearby cells, are necessarily physically clustered on disk, each pair of objects (P, Q) may require a page read, thereby causing a large number of I/O's. The point here is that if object-by-object distance calculations are to be done exactly as described in step 5(c)(3), then a page may be read many times.

The above scenario is an extreme case whereby no tuples/pages are stored in main memory. A natural question to ask is if, in the object-pairwise phase, it is possible to read each page only once. Let $Page(C)$ denote the set of pages that store at least one tuple mapped to cell C. Then for a particular *white* cell C_w, we need to read the pages in $Page(C_w)$. Because we also need the tuples mapped to a cell C_v that is an L_2 neighbour of C_w (i.e., $C_v \in L_2(C_w)$), we need the pages in $Page(L_2(C_w)) = \bigcup_{C_v \in L_2(C_w)} Page(C_v)$. Furthermore, if we want to guarantee that pages in $Page(L_2(C_w))$ are only read once, we need to read the pages: (i) that are needed by C_v, because C_v itself may be a *white* cell, and (ii) that use C_v, because C_v can be an L_2 neighbour of many other *white* cells. In general, the "transitive closure" of this kind of page cascading may cover every page of the dataset. In other words, the only way to guarantee that a page is read at most once in the object-pairwise phase is to have a buffer the size of the dataset, which is clearly a strong assumption for large datasets.

Our approach is a "middle ground" scenario whereby only a selected subset of tuples are kept in main memory. The selected subset is the set of all tuples mapped to *white* cells. Hereinafter, we refer to such tuples as *white tuples*. This is our choice partly because these are the very tuples that need object-by-object calculations, and partly because the number of tuples in a *white* cell, by definition, is bounded above by M. Furthermore, we classify all pages into three categories:

A. Pages that contain some *white* tuple(s), but may also contain some *non-white* tuples
B. Pages that do not contain any *white* tuple, but contain tuple(s) mapped to a *non-white* cell which is an L_2 neighbour of some white cell
C. All other pages

To minimize page reads, our algorithm first reads Class A pages, and then Class B pages. Following this, it suffices to re-read Class A pages to complete the object-pairwise phase. Class C pages are not needed in the object-pairwise phase.

Consider a simple example where there are 600 pages in a dataset. Suppose pages 1 to 200 are Class A pages, pages 201 to 400 are Class B pages, and pages 401 to 600 are Class C pages. Suppose tuple P is mapped to *white* cell C_w, and is stored in (Class A) page i. For P to complete its object-by-object distance calculations, these three kinds of tuples are needed:

1. *white* tuples Q that are mapped to a *white* L_2 neighbour of C_w
2. *non-white* tuples Q mapped to a *non-white* L_2 neighbour of C_w, and stored in page $j \geq i$
3. *non-white* tuples Q mapped to a *non-white* L_2 neighbour of C_w, but stored in page $j < i$

For the first kind of tuples, the pair (P, Q) is kept in main memory after the first 200 pages have been read, because both tuples are *white*. Thus, their distance can be computed and the right counters (i.e., both P's and Q's) may be updated after all Class A pages have been read. For the second kind of tuples, the distance between the pair (P, Q) can be processed when page j is read into main memory, because P is already in main memory by then since $i \leq j$. The fact that Q is not kept around afterwards does not affect P at all. Thus, after the first 400 pages have been read (i.e., all class A and B pages), the second kind of tuples for P have been checked. The only problem concerns the third kind of tuples. In this case, when Q (which is stored in page j) was read into main memory, P (which is stored in page $i > j$) had not yet been read. Since Q is a *non-white* tuple and was not kept around, then when P eventually became available, Q was gone. To deal with this situation, page j needs to be re-read. In general, all Class A pages (except one) may need to be re-read. But it should be clear that because all *white* tuples are kept in main memory, it is sufficient to read Class A pages a second time.

Before we present the formal algorithm, we offer two generalizations to the above example. First, the above example assumes that all Class A pages precede all Class B pages in page numbering, and that pages are read in ascending order. Our argument above applies equally well if these assumptions are not made—so long as all Class A pages are read first, followed by all Class B pages, and the necessary (re-reading of) Class A pages. Second, Class A pages can be further divided into two subclasses: (A.1) pages that do not store any *non-white* tuple that is needed, and (A.2) pages that store some *non-white* tuple(s) that are needed. If this subdivision is made, it should be obvious from the above analysis that in re-reading all Class A pages, it suffices to re-read only the Class A.2 pages. For simplicity, this optimization is not described in the algorithm below.

5.2 Algorithm FindAllOutsD for disk-resident datasets

Figure 3 presents algorithm FindAllOutsD for mining outliers in large disk-resident datasets. Much of the processing in the first 5 steps of algorithm FindAllOutsD is similar to that described for algorithm FindAllOutsM shown in Fig. 2. We draw attention to step 2 of algorithm FindAllOutsD, which no longer stores P but makes a note of the fact that P's page contains some tuple(s) mapped to C_q. This is important because (i) we may need to access a given cell's tuples later in the algorithm, and (ii) we need to know which cells have tuples from a particular page.

Step 5(c)(2) colours a *white* cell *yellow* if it has been determined that every tuple in a given cell is an outlier. Its tuples will be identified in step 8 after they have been read from their pages in step 6. Step 6 reads only those pages containing at least one *white* or *yellow* tuple. With respect to Sect. 5.1, this corresponds to reading all Class A pages. The *white* and *yellow* tuples from these pages are stored with the cell C_w to which they have been quantized. C_w stores exactly $Count_w$ tuples, and this count is $\leq M$. To prepare for L_2 processing, step 6(b)(1)(ii) initializes the D-neighbour counter to the number of tuples in $C_w \cup L_1(C_w)$.

In step 7, for each non-empty *white* cell C_w, we determine how many more D-neighbours each tuple $P \in C_w$ has, using (as potential neighbours) just the tuples which were read and stored in step 6. As soon as we find that P has $> M$ D-neighbours, we mark P as a non-outlier. After step 7, it is possible that some (or even all) of the non-empty *white* cells will not require any more pages, thereby reducing the number of reads in step 9.

Necessary disk reads for cells that are both *non-white* and *non-yellow* are performed in step 9. With respect to Sect. 5.1, this corresponds to reading all Class B pages, and re-reading (some) Class A pages. Again, we determine how many more D-neighbours that each tuple P in each *white* cell has, using only the newly read tuples

Algorithm FindAllOutsD

1. For $q \leftarrow 1, 2, \ldots m,\quad Count_q \leftarrow 0$
2. For each object P in the dataset, do:
   a. Map P to its appropriate cell C_q but do not store P.
   b. Increment $Count_q$ by 1.
   c. Note that C_q references P's page.
3. For $q \leftarrow 1, 2, \ldots, m,\quad$ if $Count_q > M$, label C_q *red*.
4. For each *red* cell C_r, label each of the L_1 neighbours of C_r *pink*, provided the neighbour has not already been labelled *red*.
5. For each non-empty *white* (i.e., uncoloured) cell C_w, do:
   a. $Count_{w2} \leftarrow Count_w + \sum_{i \in L_1(C_w)} Count_i$
   b. If $Count_{w2} > M$, label C_w *pink*.
   c. else
      1. $Count_{w3} \leftarrow Count_{w2} + \sum_{i \in L_2(C_w)} Count_i$
      2. If $Count_{w3} \leq M$, label C_w *yellow* to indicate that all tuples mapping to C_w are outliers.
      3. else $Sum_w \leftarrow Count_{w2}$
6. For each page i containing at least 1 *white* or *yellow* tuple, do:
   a. Read page i.
   b. For each *white* or *yellow* cell C_q having tuples in page i, do:
      1. For each object P in page i mapped to C_q, do:
         i. Store P in C_q.
         ii. $Kount_P \leftarrow Sum_q$
7. For each object P in each non-empty *white* cell C_w, do:
   a. For each *white* or *yellow* cell $C_L \in L_2(C_w)$, do:
      1. For each object $Q \in C_L$, if $dist(P, Q) \leq D$:
         Increment $Kount_P$ by 1. If $Kount_P > M$, mark P as a non-outlier, and goto next P.
8. For each object Q in each *yellow* cell, report Q as an outlier.
9. For each page i containing at least 1 tuple that (i) is both *non-white* and *non-yellow*, and (ii) maps to an L_2 neighbour of some white cell C, do:
   a. Read page i.
   b. For each cell $C_q \in L_2(C)$ that is both *non-white* and *non-yellow*, and has tuples in page i, do:
      1. For each object Q in page i mapped to C_q, do:
         i. For each non-empty *white* cell $C_w \in L_2(C_q)$, do:
            For each object $P \in C_w$, if $dist(P, Q) \leq D$:
            Increment $Kount_P$ by 1. If $Kount_P > M$, mark P as a non-outlier.
10. For each object P in each non-empty *white* cell, if P has not been marked as a non-outlier, then report P as an outlier.

Fig. 3. Pseudo-code for algorithm FindAllOutsD

from disk. If P has $> M$ D-neighbours, then P is marked as a non-outlier, and no further comparisons involving P are necessary.

5.3 Analysis of algorithm FindAllOutsD and comparison with algorithm NL

Algorithm FindAllOutsD has a linear complexity with respect to N for the same reasons explained for algorithm FindAllOutsM (cf: Sect. 4.3), but by design, algorithm FindAllOutsD has the following important advantage over algorithm FindAllOutsM with respect to I/O performance.

Lemma 4. Algorithm FindAllOutsD requires at most 3 passes over the dataset.

Proof outline: The initial-mapping phase requires one pass over the dataset. Let n be the total number of pages in the dataset. Then if n_1, n_2, and n_3 denote the number of pages in Classes A, B, and C, respectively (cf: Sect. 5.1), it is necessary that $n = n_1 + n_2 + n_3$. As shown in Sect. 5.1, the maximum total number of pages read in the object-pairwise phase is given by $n_1 + n_2 + n_1$, which is obviously $\leq 2n$. Thus, the entire algorithm requires no more than 3 passes. □

Note that the above guarantee is conservative for two reasons. First, the sum $n_1+n_2+n_1$ can be smaller than n. For example, if $n_1 \leq n_3$, then the sum is $\leq n$, implying that while some page may be read 3 times, the total number of pages read is equivalent to (no more than) 2 passes over the dataset. Second, the above guarantee assumes that: (i) there is enough buffer space for storing the *white* tuples (as will be shown later, this is not a strong assumption because typically there are not too many non-empty *white* cells), and (ii) there is only one page remaining in the buffer space for Class A and B pages. More buffer space can be dedicated to keep more Class A pages around, thereby reducing the number of page re-reads for Class A pages.

At this point, let us revisit algorithm NL, used for block-oriented, nested-loop processing of disk-resident datasets (cf: Sect. 3.2). We will show that algorithm FindAllOutsD guarantees a lower number of dataset passes (in the general case) than algorithm NL does, for sufficiently large datasets.

Lemma 5. If a dataset is divided into $n = \lceil \frac{100}{B} \rceil$ blocks, then (i) an upper bound on the total number of block reads required by algorithm NL is $n + (n-2)(n-1)$, and (ii) the number of passes over the dataset is $\geq n-2$.

Proof outline: (i) Each of the n blocks must be read exactly once during the first dataset pass. At the end of each pass, we retain 2 blocks in memory, so only $n-2$ additional blocks need to be read during passes $2, 3, \ldots, n$. Thus, $n + (n-2)(n-1)$ blocks are read. (ii) The number of dataset passes is: $\frac{n+(n-2)(n-1)}{n} = \frac{n^2-2n+2}{n} = n - 2 + \frac{2}{n} \geq n-2$. □

In general, algorithm NL may require many more passes than algorithm FindAllOutsD. For example, if a large dataset is split into $n = 10$ pieces, Lemma 5 states that algorithm NL may need $n - 2 = 10 - 2 = 8$ passes, which is 5 more passes than algorithm FindAllOutsD may need.

6 Empirical behaviour of the algorithms

6.1 Experimental setup

Our base dataset is an 855-record dataset consisting of 1995-96 NHL player performance statistics. These statistics include numbers of goals, assists, points, penalty minutes, shots on goal, games played, power play goals, etc. Since this real-life dataset is quite small, and since we want to test our algorithms on large, disk-resident datasets, we created a number of synthetic datasets mirroring the distribution of statistics within the NHL dataset. More specifically, we determined the distribution of the attributes in the original dataset by using a 10-partition histogram. Then, we generated datasets containing between 10,000 and 2,000,000 tuples, and whose distribution mirrored that of the base dataset. Each page holds up to 13 tuples.

All of our tests were run on a Sun Microsystems UltraSPARC-1 machine having 128 MB of main memory. Unless otherwise indicated, all times shown in this paper are CPU times plus I/O times. In cases where the CPU timer exceeded 2147 seconds, we use the ">" operator to denote that the time presented is a lower bound.[5] Our code was written in C++ and was processed by an optimizing compiler. The modes of operation that we used, and their acronyms, are as follows:

1. CS, which is a multidimensional cell structure implementation as described by either algorithm FindAllOutsM or FindAllOutsD. The context makes it clear which algorithm is being evaluated.
2. NL, which is an implementation of algorithm NL. The amount of memory permitted for buffering in each NL case is the same amount of memory required for the CS implementation. For example, if CS uses 10 MB of main memory, then 10 MB is also available for NL.
3. KD, which is a memory-based k-d tree implementation.
4. RT, which is a disk-based R-tree range query implementation.

Range query processing in KD and RT modes is optimized to terminate as soon as the number of D-neighbours exceeds M.

6.2 Varying the dataset size

The graph in Fig. 4 shows results for various modes and various dataset sizes for 3-D, using $p = 0.9999$. Specifically, it shows how CPU + I/O time is affected by the

[5] Our timer wraps around after 2147 seconds; hence, times beyond this are unreliable. Where we have quoted a CPU + I/O figure > 2147, it is because the CPU time was ≤ 2147, but the I/O time actually caused the sum to exceed 2147.

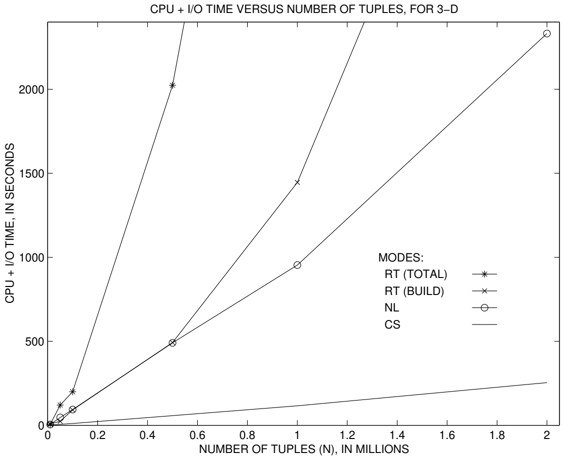

Fig. 4. How CPU+I/O time scales with N for 3-D disk-resident datasets, using $p = 0.9999$

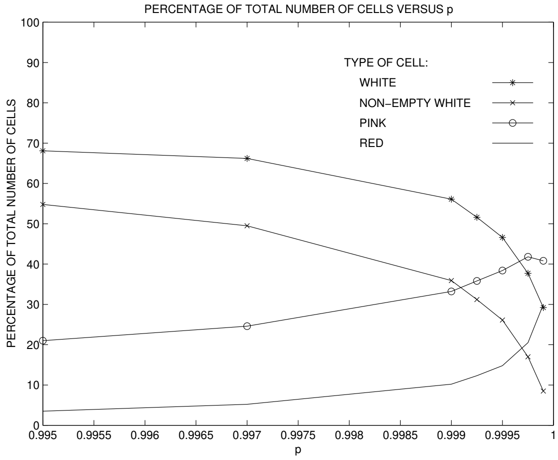

Fig. 5. 3-D cell colouring statistics for variable p, for 500,000 tuples

number of tuples. For example, CS takes 254 seconds in total time to find all the appropriate DB-outliers in a 2 million tuple dataset. In contrast, NL takes 2332 seconds, almost 10 times as long. RT mode is even less competitive. Unlike CS and NL, RT is not linear with respect to N. In fact, just building the R-tree can take at least 10 times as long as CS, let alone searching the R-tree.

While the preceding paragraph concerns disk-resident datasets, the following table summarizes the results for *memory-resident* datasets. The units are CPU + I/O seconds.

N	CS	NL	KD
20000	0.32	1.02	3.14
40000	0.54	4.26	20.49
60000	0.74	9.64	33.08
80000	1.04	17.58	54.66
100000	1.43	27.67	104.28

Again, CS can outperform NL by an order of magnitude, and the index-based algorithm—a k-d tree (KD) in this case—takes much longer, even if we just consider the time it takes to build the index.

6.3 Varying the value of p

Although Fig. 4 shows results for $p = 0.9999$ only, other values of p produce similar results. Figure 5 shows how the percentages of *white*, *pink*, *red*, and non-empty *white* cells vary with the value of p.[6] The total number of cells is simply the sum of the *red, pink,* and *white* cells. Processing time is less when there is a greater percentage of *red* and *pink* cells for the simple reason that we can generally eliminate a larger number of tuples from being considered as outliers. The success of the cell-based algorithms is largely due to the fact that many of the cells may be *red* or *pink*, and that there may be relatively few non-empty *white* cells. Recall that non-empty *white* cells require the most computational effort.

[6] We include *yellow* cells with the *white* cell population since *yellow* cells are just a special type of *white* cell.

6.4 Varying the number of dimensions and cells

In this section, we see how the number of dimensions and cells affects performance. We omit the trivial 1-D and 2-D results, but show results for 3-D, 4-D, and 5-D. Beyond 5-D, we believe that NL will be the clear winner. Table 1 shows how CS and NL compare in different dimensions for disk-resident datasets of various sizes, using $p = 0.9999$.

For CS mode in 5-D, we varied the number of partitions m_i in a given dimension i. We chose $m_i = 10$, 8, and 6 for each dimension. Thus, the columns CS(10^5), CS(8^5), and CS(6^5) stand for cases where the cell structure contains 10^5, 8^5, and 6^5 cells, respectively. The table shows that: (i) CS clearly outperforms NL in 3-D, (ii) CS often outperforms NL in 4-D,[7] and (iii) NL is the clear winner in 5-D. It is interesting to note that CS is not even competitive with NL in 5-D. Reducing the number of cells in each dimension helps CS, but is still not sufficient to catch up with NL. The key reason is the exponential growth of cells in CS. As a side effect, because we made CS and NL use the same amount of memory for fair comparison, the amount of buffer space that was available to NL increased as the number of dimensions increased. This explains why the execution time of NL shown in the above table dropped with increasing dimensions. If the buffer space had been kept constant, the execution time would have increased.

To prove the relative efficiency of CS in 3-D and 4-D for large, disk-resident datasets (e.g., N=2,000,000), it is worth noting that *even if 100% buffering were permitted for NL*, CS may still outperform NL, as shown by the following table. All times are total times, and the units are seconds.

[7] We note that NL sometimes beats CS in 4-D. We ran many other tests using different values of p and using datasets of various sizes, and most of the time, CS was the clear winner. In those cases where CS did not win, the margin of victory for NL was almost always small (e.g., 10% or less). The margin of victory does not appear to be dependent on p since both p close to unity and p further away from unity result in many significant victories for CS.

Table 1. Comparison of CS and NL algorithms, when $p = 0.9999$

	3-D		4-D		5-D			
N	CS(10^3)	NL	CS(10^4)	NL	CS(10^5)	CS(8^5)	CS(6^5)	NL
100,000	13.26	77.58	42.99	45.79	513.78	364.63	185.19	17.37
500,000	68.44	490.39	234.77	223.51	>2147	1960.95	1061.33	120.22
2,000,000	281.91	2159.42	1042.43	1421.16	>2147	>2147	>2147	1555.78

Mode	$p = 0.9999$	0.99993	0.99996	0.99999
CS (3-D)	253.90	230.71	210.25	182.90
NL (3-D)	703.40	534.65	367.40	211.97
CS (4-D)	606.56	550.20	396.52	246.79
NL (4-D)	965.60	720.23	477.92	247.93

In 3-D, the memory used was 47 MB for NL and 9 MB for CS. CS only performed between 1.04 and 1.33 dataset passes for each of the 3-D cases, which is well below the maximum of 3 set by Lemma 4. In 4-D, the memory used was 55 MB for NL and 21 MB for CS. CS only performed between 1.09 and 1.44 dataset passes for each of these cases. Even though CS performs more reads than NL (with 100% buffering), CS's performance is superior because it performs far fewer object-to-object comparisons.

7 Case studies of distance-based outliers

Thus far, we have focused on showing the first objective of this paper, that is, that distance-based outliers can be detected efficiently for large datasets, or for k-D datasets with large values of k. In this section, we focus on showing that distance-based outliers are meaningful semantically, and admit applications that are currently not supported by other notions of outliers.

To do so, we present anecdotal evidence of three separate case studies of different flavours. The first one corresponds to a publicly accessible dataset of NHL hockey players' statistics. It has 16 attributes, most of which are numeric.

The second application is a prototype jointly developed with Point Grey Research, which sells video surveillance cameras that can track the motion of each person within the field of view of the cameras. Our specific prototype extracts a 2-D trajectory of each person and identifies persons whose trajectories are abnormal. This represents an interesting case study of mining video data for exceptional spatio-temporal behaviour.

The third study is an evaluation of employers' performance on injured workers' compensation. This was joint work with the Workers' Compensation Board of the province of British Columbia. The evaluation, which lasted for four months, was completed in November, 1998. The reason why this application is interesting is that it included a complete evaluation by non-technical domain experts judging the value of the detected outliers.

7.1 NHL players' statistics

NHL players' statistics can be obtained from Web sites such as nhlstatistics.hypermart.net. Typically, a dataset of a certain year (i.e., an "NHL season") contains a tuple for every player who appeared in some NHL game during that year. Each tuple describes such information as games played, goals scored, penalty minutes accumulated, number of shots on goal taken, and so forth.

During in-lab experiments on this kind of NHL data, we have identified outliers among players having perhaps "ordinary looking" statistics which suddenly stand out as being non-ordinary when combined with other attributes. Portions of sample runs of these experiments are documented in Fig. 6.

The first example shows that, in 1994, Wayne Gretzky and Sergei Fedorov were outliers when 3 attributes—points scored, plus-minus statistic,[8] and number of penalty minutes—were used. Fedorov was an outlier because his point and plus-minus figures were so much higher than almost all other players. (As a reference note, the "average" NHL player has fewer than 20 points, a plus-minus statistic of zero, and fewer than 100 penalty minutes.) Gretzky was an outlier predominantly because of his high point total and low plus-minus figure. In fact, we were surprised that Gretzky's plus-minus figure was so poor, especially since he was the highest scorer in the league that year, and since high scorers usually have positive plus-minus values (as confirmed by Fedorov in the same example).[9] When the same 3 attributes were used for 1996 data (see the second example), Vladimir Konstantinov had an astonishingly high plus-minus statistic (i.e., +60) despite the fact that his point total was rather mediocre.

Our third example shows that Chris Osgood and Mario Lemieux were outliers when 3 attributes—games played, goals scored, and shooting percentage (i.e., goals scored, divided by shots taken)—were used. Few NHL players had shooting percentages much beyond 20%, but Osgood's shooting percentage really stood out. Despite playing 50 games, he only took 1 shot, on which he scored. Osgood's outlier status is explained by the fact

[8] The plus-minus statistic indicates how many more even-strength goals were scored by the player's team—as opposed to the opposition's team—when this particular player was on the ice. For example, a plus-minus statistic of +2 indicates that this player was on the ice for 2 more goals scored *by* his team than *against* his team.

[9] The next highest scorer in negative double digits did not occur until the 23rd position overall. Perhaps Gretzky's plus-minus can be explained by the fact that he played for the Los Angeles Kings that year—a team not known for its strong defensive play.

```
FindOutliers  nhl94.data  p=0.998  D=29.6985  POINTS  PLUSMINUS  PENALTY_MINUTES
  1)  Name = WAYNE GRETZKY,  POINTS = 130, PLUSMINUS = -25, PENALTY_MINUTES = 20
  2)  Name = SERGEI FEDOROV, POINTS = 120, PLUSMINUS =  48, PENALTY_MINUTES = 34

FindOutliers  nhl96.data  p=0.998  D=26.3044  POINTS  PLUSMINUS  PENALTY_MINUTES
  1)  Name = VLAD KONSTANTINOV, POINTS =  34, PLUSMINUS =  60, PENALTY_MINUTES = 139

FindOutliers  nhl96.data  p=0.997  D=5  GAMES_PLAYED  GOALS  SHOOTING_PERCENTAGE
  1)  Name = CHRIS OSGOOD,  GAMES_PLAYED = 50, GOALS =  1, SHOOTING_PERCENTAGE = 100.0
  2)  Name = MARIO LEMIEUX, GAMES_PLAYED = 70, GOALS = 69, SHOOTING_PERCENTAGE =  20.4

FindOutliers  nhl96.normalized.data  p=0.996  D=0.447214  GAMES_PLAYED  POWER_PLAY_GOALS
              SHORTHANDED_GOALS  GAME_WINNING_GOALS  GAME_TYING_GOALS
  1)  Name = ALEXANDER MOGILNY, GAMES_PLAYED = 79, POWER_PLAY_GOALS = 10,
              SHORTHANDED_GOALS = 5, GAME_WINNING_GOALS = 6, GAME_TYING_GOALS = 3
  2)  Name = MARIO LEMIEUX, GAMES_PLAYED = 70, POWER_PLAY_GOALS = 31,
              SHORTHANDED_GOALS = 8, GAME_WINNING_GOALS = 8, GAME_TYING_GOALS = 0
```

Fig. 6. Sample output involving NHL players' statistics

that he is a goalie, and that goalies have rarely scored in the history of the NHL. Lemieux was an outlier, not because he scored on 20.4% of his shots, but because no other player in the 20% shooting range had anywhere near the same number of goals and games played.

Our fourth example shows outliers in higher dimensions. To begin our discussion, we point out that none of Alexander Mogilny's statistics was extreme in any of the 5 dimensions. In contrast, in the same example, 3 of 5 statistics for Mario Lemieux were extreme. The NHL range for each of the 5 attributes (with Mogilny's statistic in parentheses) was as follows: 1–84 games played (79), 0–31 power play goals (10), 0–8 shorthanded goals (5), 0–12 game winning goals (6), and 0–4 game tying goals (3). The significance of this example is that Mogilny possesses certain skills that are unlike those of other players in the league. Specifically, he is an excellent "special teams" player because of his well above average contributions in *numerous* goal-scoring categories, not just in one or two such coveted categories.

The Mogilny example brings out an interesting point. In some of the statistics literature (e.g., [31]), multivariate outliers are classified into two forms: *gross* outliers and *structural* outliers. Gross outliers are those observations that are outlying for one or more individual attributes. In other words, a gross outlier is an outlier in one dimension for at least one variable. Structural outliers do not share this property, and may or may not be detected visually in 2-D scatterplots or in 3-D spin plots. In some cases, they may be identified only when all k dimensions are simultaneously considered. Mogilny is an example of a structural outlier which would not be able to be detected using visual means.

In professional sports, general managers are often looking for a competitive edge in constructing a championship team, especially when considering: (i) the team owner's huge investment in players' salaries (e.g., \$50 million annually), (ii) monetary constraints on future spending, and (iii) the existence of a plethora of player performance statistics, making it difficult to routinely compare one player with another. For example, a general manager might be able to acquire outlying (and perhaps relatively inexpensive) players that have unusual, but desirable, playing characteristics. While past performance is no guarantee of future performance, an NHL general manager who is aware of Mogilny's unique skill set with respect to special teams play, for example, may wish to make him a part of the team. Mogilny's agent, on the other hand, may use such information to justify a salary increase with his current or future team.

In all of the examples shown in this section, the user chose suitable values for p and D to define the "strength" of the outliers requested. As mentioned in Sect. 4.6, sampling techniques can be used to estimate a default value of D for a given value of p. After the initial run, the user can reset both p and D.

7.2 Spatio-temporal trajectories from surveillance videos

7.2.1 Motivation and significance

Video surveillance is a powerful method of helping to ensure the safety of public places. While there are millions of cameras videotaping public places, automated systems that can intelligently monitor a scene are few and far between. The material below reports on the development of an automated system that is able to identify suspicious behavior of people in public places. The task at hand is the motion tracking of people walking through an open space such as a building atrium or a large hallway. People generally take similar paths as they are walking from point A to point B in a building. It is up to the system to identify paths that are unusual, and to bring such paths to the attention of the human user.

From the standpoint of knowledge discovery from databases, the study reported here could be of general interest in the following aspects:

- Video/image data mining
  Over the years, huge repositories of image data (e.g., satellite images) and video data (e.g., sensor or surveillance data) have been created. While they represent a vast and valuable source for knowledge discovery, not many techniques have been developed to carry

out the task. Given the non-traditional nature of the data (i.e., raster data), one key complication is finding a suitable representation of the data. Our approach is to extract and abstract 2-D (it could easily be 3-D) path trajectories from the video data as an intermediate representation of the data.
- Spatio-temporal analysis
  Spatio-temporal analysis is important for many applications, such as for finding the migration patterns of diseases. Again, there is the issue of representing the spatio-temporal data, which need not be video data and may simply be alphanumeric in nature. Our approach of representing the data as 2-D paths appears to be a natural way for carrying out the analysis.
- Finding exceptions
  In many surveillance and sensor style applications, the majority of patterns are often not as interesting and valuable as the exceptions. Our approach of finding distance-based outliers/exceptions appears to be appropriate and effective.

Motion tracking is an active area of research in computer vision. There are a number of techniques used for tracking people [14, 16, 19]. The specific technique for motion tracking which we use is closely related to the work discussed in [11].

To the best of our knowledge, very few techniques have been developed for video/image and spatio-temporal analysis. Work performed by researchers at the Jet Propulsion Laboratory has perhaps been the most visible such work. Stolorz, et al. [29] developed the CONQUEST system which provides a querying and exploratory data analysis environment for geoscientific image data. In terms of its methodology, CONQUEST relies on cluster analysis and massively parallel feature extraction. Our methodology relies on 2-D path trajectories and outlier detection. Stolorz, et al. [30] also developed the Quakefinder system which automatically monitors tectonic activity in the earth's crust by examining satellite data. The technical focus is on certain kinds of statistical inference, massively parallel computing, and global optimization. To locate volcanos on Venus, Burl, et al. [7] developed the JARtool. The approach taken is based on matched filtering, followed by feature extraction assisted by training data, and then classification learning. See [12] for a more comprehensive discussion.

7.2.2 Overview of the prototype

There are two main modules of the prototype. First, there is the module that produces the spatio-temporal trajectories from surveillance videos. This is mainly computer vision work done by Point Grey Research. The only reason why this material is included in this paper is to give a complete picture of the processing done by the prototype. Second, there is the module that computes the exceptional trajectories. This part is directly due to distance-based outliers and is the main focus below.

Trajectory Extraction
The module consists of a stereo vision system connected

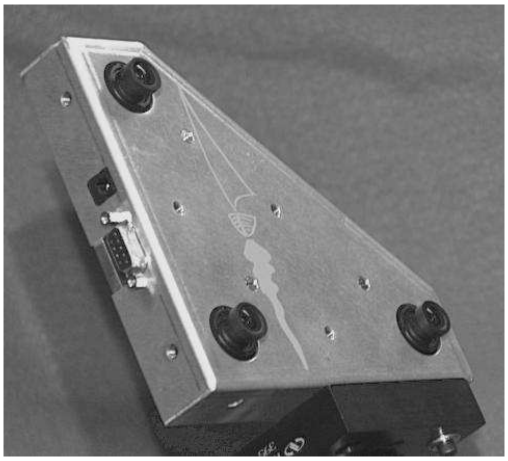

Fig. 7. Triclops stereo vision system

to a standard personal computer. Figure 7 shows the three-camera module of the Triclops stereo vision system marketed by Point Grey Research. The Triclops stereo vision system produces distance measurements for each pixel of the grey-scale images. The basic principle behind the stereo vision technology is to establish a correspondence between images obtained from slightly offset cameras. The benefits of using stereo vision technology are that it is robust to changes in the lighting condition of the environment, and that it produces a continuous and full field of 3-D results.

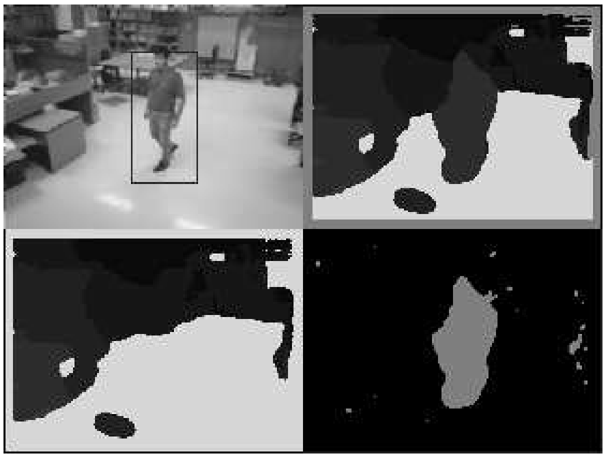

Fig. 8. Tracking a person using the Triclops stereo vision system

Figure 8 shows the output of the Triclops system along with the results of the people-tracking program. (In general, the system is able to separate and track multiple people in the scene.) The top left image in Fig. 8 is the grey-scale image from the Triclops system with a bounding box identifying the position of the person in the scene. The top right image is the current depth map of the scene, and the bottom left image is the depth image of the background. Finally, the bottom right image is the difference between the background and the current depth map. This image is used in order to identify where the person is.

The people-tracking program assumes that there is only one person walking through the scene, and that the background is static. In order to track the person, the depth image of the background is obtained. The difference indicates that there has been a change in the scene. The largest "blob" in the scene is assumed to be the person, and the 3-D position of the centroid of the "blob" relative to the camera is established. This 3-D position is then translated into the 2-D coordinate system of the floor plane.

The 2-D coordinates of the person along with a timestamp are stored in a file for future analysis. Each time a person walks into the scene, a new file is created and information is stored until the person leaves the scene.

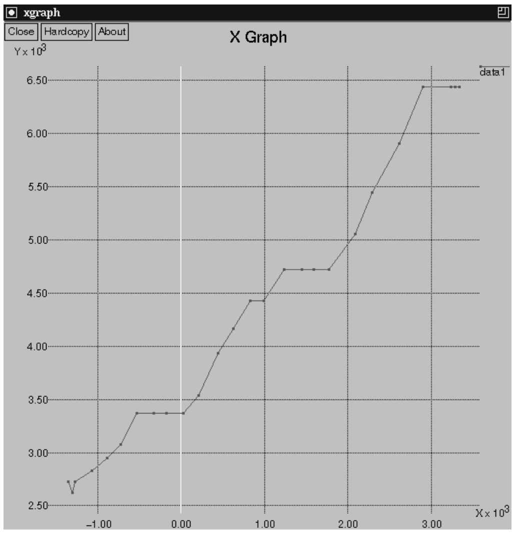

Fig. 9. Path produced by the stereo vision system

Figure 9 shows the 2-D trajectory made by the person in Fig. 8. The coordinate system represents the plane in which the person is moving, not the image plane. In other words, the system is capable of reconstructing the 3-D path of the person, which is then transformed into a 2-D trajectory.

Distance functions for trajectories

To detect distance-based outliers, we need to define a distance function. For surveillance applications, we can think of many distance measures that can be useful. Below, we focus on two particular ones.

As pointed out before, a 2-D trajectory can be represented in terms of timestamped 2-D points. This representation, however, may be too detailed in the comparison between paths, and may require too much storage. Furthermore, there are complications due to paths of different lengths and densities. In order to capture the key features of each trajectory, the trajectories are summarized by the following features:

- Start and end points: the x, y coordinates of the points where the person enters and leaves the field of view
- Number of points: effectively the length of the trajectory
- Heading: the average, minimum, and maximum values of the directional vector of the tangent of the trajectory at each point
- Velocity: average, minimum, and maximum velocity of the person during the trajectory

Summarizing the trajectory of a person's path in this fashion provides a multidimensional space in which paths can be compared. More formally, the definition of a path can be written as:

$$P = \begin{bmatrix} P_{\text{start}} \\ P_{\text{end}} \\ P_{\text{heading}} \\ P_{\text{velocity}} \end{bmatrix}$$

where:
$P_{\text{start}} = (x_{\text{start}}, y_{\text{start}})$,
$P_{\text{end}} = (x_{\text{end}}, y_{\text{end}})$,
$P_{\text{heading}} = (avg_{\text{heading}}, max_{\text{heading}}, min_{\text{heading}})$,
$P_{\text{velocity}} = (avg_{\text{velocity}}, max_{\text{velocity}}, min_{\text{velocity}})$.

We experimented with the following distance function between two paths P_1 and P_2. Essentially, it is a weighted sum of differences along the various dimensions:

$$D(P_1, P_2) =$$

$$\begin{bmatrix} D_{\text{start}}(P_1, P_2) \\ D_{\text{end}}(P_1, P_2) \\ D_{\text{heading}}(P_1, P_2) \\ D_{\text{velocity}}(P_1, P_2) \end{bmatrix} * \begin{bmatrix} w_{\text{start}} & w_{\text{end}} & w_{\text{heading}} & w_{\text{velocity}} \end{bmatrix}$$

This distance function can be quite effective, with appropriate weights. The weights were determined by domain experts.

Another distance function which we experimented with is:

$$D_{\text{feature}}(P_1, P_2) =$$

$$\begin{cases} D_{\text{euclid}}(P_{1,\text{feature}}, P_{2,\text{feature}}) \\ \qquad \text{if } D_{\text{euclid}}(P_{1,\text{feature}}, P_{2,\text{feature}}) < Thres_{\text{feature}} \\ \infty \quad \text{otherwise} \end{cases}$$

such that D_{euclid} represents the Euclidean distance and $Thres_{\text{feature}}$ is defined as:

$$Thres_{\text{feature}} = \lambda * (max(T_{\text{feature}}) - min(T_{\text{feature}}))$$

where λ is a global parameter for all dimensions, and T_{feature} is an array of feature points in the database of paths. Operators min and max select the minimum and maximum values of the feature, respectively. The basic intuition behind this approach is that an outlier in a multidimensional space must be significantly far away from the rest of data points along at least one dimension. The intuitive explanation of parameter λ is that it specifies the fraction of the distance between two furthest points in each dimension. This distance function was found to be quite effective empirically. The following experimental results were obtained using this distance function.

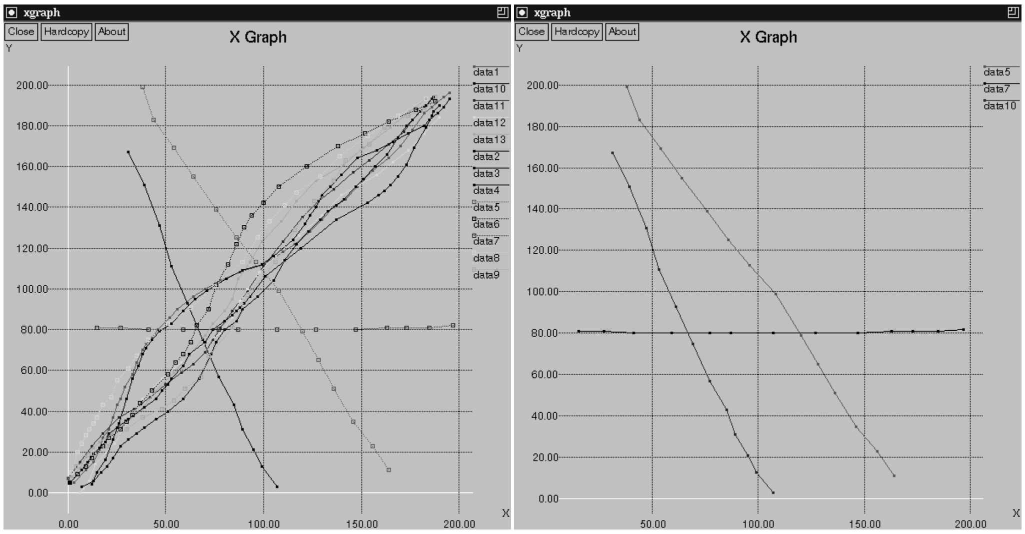

Fig. 10. Finding outliers based on entry and exit points. The left figure shows the entire dataset; the right figure shows the outliers only.

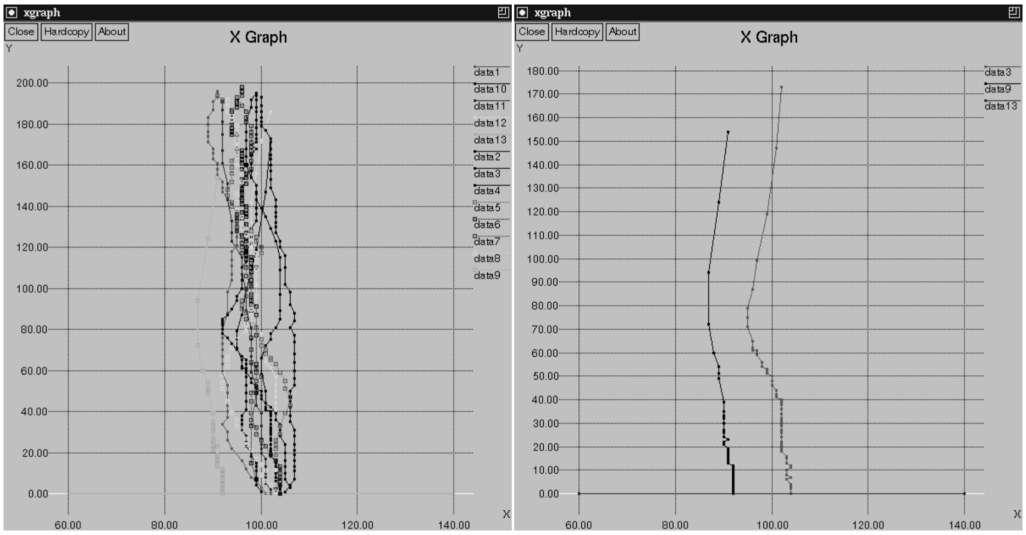

Fig. 11. Finding outliers based on speed of motion. The left figure shows the entire dataset; the right figure shows the outliers only.

7.2.3 Preliminary experimental results

Figure 10 presents a small dataset[10] with a number of trajectories going from the bottom-left to the top-right of the coordinate system. The dataset also contains trajectories that do not share an entry or an exit point. These trajectories are successfully identified as outliers.

Figure 11 presents a dataset that has all paths starting and ending at approximately the same location. In this case, the velocity in all paths is not uniform. Two paths have high speed portions which can be observed by the reduced density of points in the graph. These trajectories were found as outliers.

Finally, Fig. 12 shows a dataset that contains outliers due to differences in speed and/or trajectories between the entry and exit points. Those paths are identified as outliers.

In conclusion, the video surveillance example described in this section shows how the notion of DB-outliers can be successfully used to find exceptions in spatio-temporal data.

7.3 Workers' Compensation Board employer performance data

At the Workers' Compensation Board (WCB) of British Columbia, a corporate data warehouse that stores data on 170,000 employers in the province has been made accessible to our case study. The goal is to identify employers with poor performance on injured workers' compensation (e.g., with respect to the numbers and types of claims). Traditionally, poorly performing employers are found by conventional means, such as using sorted lists containing one of several performance indicators. This primitive method may work for an employer with an extremely poor value for a chosen performance indicator; however, it cannot identify employers who are not extreme for an individual indicator, but who are nevertheless very poor for various combinations of the indicators. In other words, gross outliers (see Sect. 7.1) may be found, but structural outliers are likely to go undetected.

[10] For simplicity, the datasets in this section are very small. For this application, performance considerations are not currently an issue.

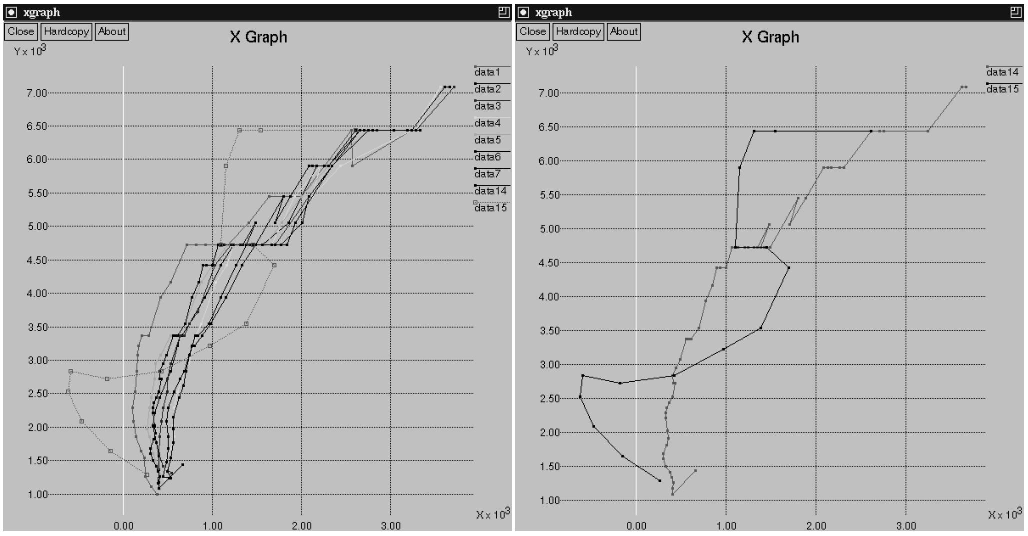

Fig. 12. Finding outliers based on speed and/or trajectory. The left figure shows the entire dataset; the right figure shows the outliers only.

Due to explicit privacy agreements made with the WCB, we are not able to go into details about the specific findings of our case study. Instead, we give a high level overview of the tasks conducted and their significance.

- Traditionally, the WCB tracks four performance indicators for each employer. We applied the distance-based outliers methodology to those indicators and identified a list of exceptional employers. This list was verified by domain experts to be valid and valuable.
- The above list of outliers includes employers with "relatively" poor performance and those with "absolutely" poor performance. From the WCB's standpoint, it is more concerned with the latter than with the former. Consequently, a new suite of performance indicators was introduced. Based on these new indicators, the list of distance-based outliers was found to be even more valuable.

For a data mining algorithm in general, there is always the issue of how much the algorithm knows about the semantics of the application domain. If the algorithm knows too little, the findings may not be as intuitive and meaningful. If the algorithm knows too much, then the algorithm may not be general enough for other application domains. While we are not claiming that it is possible to come up with a meaningful distance measure for every application, we feel that a distance measure provides a nice, succinct interface for the user to feed an otherwise syntactic algorithm with an appropriate dose of application semantics. In different analyses, a user may choose to examine the same dataset from different angles by using different distance measures. In the WCB's case, this flexibility turns out to be very useful, and because of our case study, that leads to a new understanding of the data. The WCB is now tracking the new indicators.

In closing, we point out that the WCB pays out about \$1 billion each year in claims. The results of our case study help to identify poorly performing employers, as well as the top performing employers (which can serve as models for the poor ones). There is great optimism within the WCB that this knowledge in turn will lead to a significant reduction in claims costs.

8 Conclusions

We believe that identifying DB-outliers is an important and useful data mining activity. In this paper, we proposed and analyzed several algorithms for finding DB-outliers. In addition to two simple $O(k\,N^2)$ algorithms, we developed cell-based algorithms that are linear with respect to N and are suitable for $k \leq 4$. The cell-based algorithm developed for large, disk-resident datasets also guarantees that no data page is read more than 3 times, if not once or twice. Our empirical results suggest that: (i) the cell-based algorithms are far superior to the other algorithms for $k \leq 4$ (in some cases, by at least an order of magnitude), (ii) the nested-loop algorithm is the choice for $k \geq 5$ dimensions, and (iii) finding all DB-outliers is computationally very feasible for large, multidimensional datasets (e.g., 2.5 minutes total time for 500,000 tuples in 5-D). Using algorithm NL, there is no practical limit on the size of the dataset or on the number of dimensions.

We also described our work in three real-life application domains: (i) NHL players' statistics, (ii) a video surveillance system, and (iii) workers' compensation claims. In conjunction with the performance results on the synthetic datasets described in Sect. 6, we showed that outlier detection can be done efficiently for large datasets, as well as for k-dimensional datasets with relatively large values of k (i.e., $k \geq 5$). Hopefully, these examples have served to convince the reader that distance-based outliers are meaningful semantically, and that outlier detection, in general, is an important knowledge discovery task.

Acknowledgement. We thank the anonymous referees for their constructive comments. This research has been partially sponsored by NSERC Grant OGP0138055 and NCE-IRIS grants.

References

1. Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proc ACM SIGMOD, pp 207–216
2. Arning A, Agrawal R, Raghavan P (1996) A linear method for deviation detection in large databases. In: Proc KDD, pp 164–169
3. Barnett V, Lewis T (1994) Outliers in statistical data. John Wiley & Sons
4. Bentley, JL (1975) Multidimensional binary search trees used for associative searching. In: CACM, 18(9):509–517
5. Breiman L, Friedman JH, Olshen RA, Stone, CJ (1984) Classification and regression trees. Wadsworth International Group
6. Berchtold S, Keim D, Kriegel H-P (1996) The X-tree: an index structure for high-dimensional data. In: Proc VLDB, pp 28–39
7. Burl MC, Fayyad U, Perona P, Smyth P, Burl MP (1994) Automating the hunt for volcanoes on Venus. In: Proc Conf on Vision and Pattern Recognition, pp 302–308
8. Chakrabarti S, Sarawagi S, Dom B (1998) Mining surprising patterns using temporal description length. In: Proc VLDB, pp 606–617
9. Draper NR, Smith H (1966) Applied regression analysis. John Wiley & Sons
10. Ester M, Kriegel H-P, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc KDD, pp 226–231
11. Eveland C, Konolige K, Bolles RC (1998) Background modeling for segmentation of video-rate stereo sequences. In: Proc Conf on Vision and Pattern Recognition, Santa Barbara, CA
12. Fayyad U, Haussler D, and Stolorz P (1996) KDD for science data analysis: issues and examples. In: Proc KDD, pp 50–56
13. Freedman D, Pisani R, and Purves R (1978) Statistics. WW Norton, New York
14. Gavrilla DM, Davis IS (1996) 3-D model-based tracking of humans in action: a multi-view approach. In: IEEE Computer Vision and Pattern Recognition, pp 73–80
15. Guttmann R (1984) A dynamic index structure for spatial searching. In: Proc ACM SIGMOD, pp 47–57
16. Haritaoglu I, Harwood D, Davis L (1997) Real time detection and tracking of people and their parts. University of Maryland, Technical Report, August 1997
17. Hawkins D (1980) Identification of outliers. Chapman and Hall, London
18. Hellerstein J, Koutsoupias E, Papadimitriou C (1997) On the analysis of indexing schemes. In: Proc PODS, pp 249–256
19. Isard M, Blake A (1996) Contour tracking by stochastic propagation of conditional density. In: Proc European Conf on Computer Vision, Vol 1, pp 343–356
20. Johnson T, Kwok I, Ng R (1998) Fast computation of 2-dimensional depth contours. In: Proc KDD, pp 224–228
21. Johnson RA, Wichern DW (1992) Applied multivariate statistical analysis. Third edition, Prentice-Hall
22. Knorr EM, Ng RT (1997) A unified notion of outliers: properties and computation. In: Proc KDD, pp 219–222 An extended version of this paper appears as: Knorr EM, Ng RT (1997) A unified approach for mining outliers. In: Proc CASCON, pp 236–248
23. Knorr EM, Ng RT (1998) Algorithms for mining distance-based outliers in large datasets. In: Proc VLDB, pp 392–403
24. Ng R, Han J (1994) Efficient and effective clustering methods for spatial data mining. In: Proc VLDB, pp 144–155
25. Preparata F, Shamos M (1998) Computational geometry: an introduction. Springer-Verlag
26. Ruts I, Rousseeuw P (1996) Computing depth contours of bivariate point clouds. In: Computational Statistics and Data Analysis, 23, pp 153–168
27. Samet H (1990) The design and analysis of spatial data structures. Addison-Wesley
28. Sarawagi S, Agrawal R, and Megiddo N (1998) Discovery-driven exploration of OLAP data cubes. In: Proc EDBT, pp 168–182
29. Stolorz P, Mesrobian E, Muntz RR, Shek EC, Santos JR, Yi J, Ng K, Chien SY, Nakamura H, Mechoso CR, and Farrara JD (1995) Fast spatio-temporal data mining of large geophysical datasets. In: Proc KDD, pp 300–305
30. Stolorz P, Dean C, Crippen R, and Blom R (1995) Photographing earthquakes from space. Concurrent Supercomputing Consortium Annual Report, ed. Pauna T, pp 20–22
31. White RA (1992) The detection and testing of multivariate outliers. Masters thesis, Department of Statistics, University of British Columbia
32. Zhang T, Ramakrishnan R, Livny M (1996) BIRCH: an efficient data clustering method for very large databases. In: Proc ACM SIGMOD, pp 103–114

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.